A Multi-Level Cache Model for Run-Time Optimization of Remote Visualization

Robert Sisneros, Chad Jones, Jian Huang, Jinzhu Gao, Byung-Hoon Park, and Nagiza F. Samatova

Abstract— Remote visualization is an enabling technology aiming to resolve the barrier of physical distance. While many researchers have developed innovative algorithms for remote visualization, previous work has focused little on systematically investigating optimal configurations of remote visualization architectures. In this paper, we study caching and prefetching, an important aspect of such architecture design, in order to optimize the fetch time in a remote visualization system. Unlike a processor cache or web cache, caching for remote visualization is unique and complex. Through actual experimentation and numerical simulation, we have discovered ways to systematically evaluate and search for optimal configurations of remote visualization caches under various scenarios, such as different network speeds, sizes of data for user requests, prefetch schemes, cache depletion schemes, etc. We have also designed a practical infrastructure software to adaptively optimize the caching architecture of general remote visualization systems, when a different application is started or the network condition varies. The lower bound of achievable latency discovered with our approach can aid the design of remote visualization algorithms and the selection of suitable network layouts for a remote visualization system.

Index Terms— Remote visualization, distributed visualization, performance analysis, caching.

1 Introduction

Remote visualization enables users to access visualization data stored at remote and dispersed locations without making complete local replicas.

A primary performance measure of remote visualization is the *latency* incurred during user interaction. The *absolute latency* is measured from the time a user request is generated, e.g. choosing a new view angle or a new isovalue, until the request is fulfilled. *Reducing the latency* is a challenging problem in remote visualization.

Latency hiding techniques have recently become a major approach to addressing this problem. In a nutshell, they aim to trade temporary visual quality for a steadily low response time, for instance, by intelligently using approximations of the requested data while the data is still being transmitted. In this case, user-perceived latency can be greatly reduced with the same absolute latency.

Reducing the absolute latency in remote visualization, if addressed properly, would invariably improve the overall quality of the user's experience. When latency hiding methods are not employed, then the incurred latency directly impacts the rate of client-side user interaction. With latency hiding techniques, a lower absolute latency would require fewer approximations, leading to better visual quality and accuracy during a visualization session.

A general strategy to reduce absolute latency is to prefetch data and use *caching*, similar to the use of processor and web caches. The caching mechanisms needed by remote visualization, however, are unique in several important ways (more details in Section 2). From an architectural design perspective, we are interested in the question of whether it is possible to systematically evaluate and search for optimal cache configurations for remote visualization under various scenarios, such as different network speeds, sizes of user requested data, rates of user interaction, and cache depletion schemes. If the answer is yes, we would like to leverage the knowledge learned to design a practical scheme for adaptively optimizing the caching architecture of general remote visualization systems whenever a new application is started or the overall environment adapts over time. To the best of our knowledge, this topic has not been a previous focus in the field.

In this paper, we base our study on actual experimentation as well as numerical simulation. First, we developed a network-aware caching software, called *inCache*, that runs on most major flavors of Linux. With commands sent through a TCP socket at runtime, each inCache process running on an independent network node can be individually controlled to switch to new cache configurations. Such a runtime adaptiveness enabled us to experiment with actual real-world applications in search of optimal cache configurations. Second, based on lessons learned through experiments, we further developed an efficient method of numerical simulation to discover optimized cache configurations for a remote visualization system. Although experiments often take hours to finish, numerical simulation using a single CPU completes with comparable results with sub-second efficiency. Hence, a config-

• *Corresponding author: Jian Huang, Dept. of Computer Science, University of Tennessee, Knoxville, TN 37996. Email: huangj@cs.utk.edu.*

uration with minimized fetch time under current network situation can be instantly determined and applied. From an application developer's perspective, a chain of inCache nodes orchestrated by numerical simulation could serve as a black box of prefetching and caching that transparently adapts to temporal changes in the network. As we will show in Section 6 Results , near optimal cache configurations that maximize utilization of network bandwidth can be discovered using our approach.

Another use of our findings is also interesting. That is, to attack the problem of designing latency hiding algorithms from a "negative angle." By finding the lower bound of absolute latency for remote visualization on a network, for any specific combination of targeted user request rate and average fetch size, one can predict that some algorithm will fail to hide latency effects from the user because it is impossible to receive the needed updates fast enough, while for others, there is still room to adapt for more functionality and better quality. In turn, by fixing the algorithm and user requirements, one can evaluate various ways to lay out a system on the network, and select the layout most probable to meet the requirements. For example, as shown in Section 6.1, some applications remain interactive over a wireless network, while others cannot. All such predictions can be performed without actually building the remote visualization system.

In the remainder of the paper, we discuss related work in Section 2. In Section 3, we describe our generalization of caching in remote visualization using a model of incremental updates. Our study of the overall problem space and the framework to search for an optimal configuration is presented in Sections 4 and 5, respectively. We present our results in Section 6 and then conclude with a discussion of future work in Section 7.

2 Background

Over the years, the visualization community has explored both algorithmic and architectural approaches to design remote visualization systems.

In order to design visualization algorithms suitable for use in remote settings, researchers have focused on lowering system latency by (i) reducing the amount of data needed per request to a minimum using methods based on view-dependence, level-of-detail, and compression [6, 9, 12, 13, 16, 17], or by (ii) integrating the latest graphics hardware into remote visualization systems [7, 8, 20].

Obviously, it would be ideal to have a remote visualization algorithm with a minimal level of absolute latency. When that is not feasible, however, smooth user interactions might still be achievable using latency-hiding techniques, mainly based on alternative representations of a visualization. For instance, texture mapped geometry meshes can be used to faithfully approximate volume rendered results under a small range of viewing angles [2,15]. Using such approximations of the requested data while the data is still being transmitted, user-perceived latency could sometimes be negligible. But if a user moves outside the targeted range too soon, user-perceived visual quality would then severely suffer. Indeed, latency-hiding as an algorithmic approach introduces a tradeoff between user-perceived visual quality and user-perceived latency.

Besides algorithmic approaches, general architectural aspects of remote visualization are also important venues of research. Of the several important issues to consider in architectural design, we would like to focus on the use of caching in general remote visualization systems. One particular area of study is how optimal cache configurations for all networked nodes in a remote visualization system can be systematically evaluated and discovered.

Caching, as a general concept, has been widely applied throughout computer science. We should then consider previous methods designed to optimize cache configurations for other notable applications, especially those used to optimize processor cache [4, 10] and web cache [11]. We find optimizing caches for remote visualization different from those two fields in several aspects. In particular, it differs in the nature of the problem space, the goals of optimization, and the resulting choice of the search method for seeking an optimal solution.

To optimize a processor cache, the problem space is quite uniform. All cache lines are of the same size, and identified by addresses of a constant length (e.g. 32 bits). The total number of possible cache configurations is also rather limited (for instance, 4-way associative vs. 8-way associative, under a few candidate cache-line sizes). Since hardware design does not allow dynamic variations, runtime optimization was not part of the goal. For these reasons, exhaustive search was employed to optimize processor cache configurations [4, 10].

We also compare our problem to that of web caching [11]. The focus of web caching is to create an affordable "mirror", or proxy, that provides fast access to frequently used documents while keeping data secure and fresh (i.e. updated). Also, filtering may be used to further improve performance, treating very large and very small documents differently depending on the web caching goals. A good web cache configuration is one that maximizes hit rate while serving a large number of dynamic users. The size of the problem space is not typically large. In terms of optimization, previous researchers have attempted methods such as linear programming, with an underlying assumption of a linear problem space [11].

It is also uncommon for a web cache to adapt its configuration frequently (e.g. every five seconds).

Caching in remote visualization is unique when compared to processor and web caching. Remote visualization caches are commonly designed to provide low latency interactivity to one or just a few active users. Remote visualization is data intensive, and hence needs caches to adapt to changing network bandwidth and user request rates, etc. Exacerbating the problem is a need to manage data fetches of highly varying sizes indexed by many variables. The popular use of multiple networked nodes for dispersed tasks further compounds the complexity. Also, our problem space has no linear structures to leverage (Section 4). As a result, we cannot directly apply optimization methods of processor or web caching.

Also worth noting is system configuration optimization in distributed virtual reality [21], with datasets replicated on all participating computers. The network is not the bottleneck due to minimal runtime communication (transmitting only control messages and highly compressed video streams). It is sufficient to solely optimize resource utilization on each individual PC. The optimization space is often small enough for using exhaustive search.

Finally, previous researchers have also dedicated much pioneering work towards other architectural issues, such as advanced networking protocols, scalable infrastructure of distributed systems, etc. [1,3,6]. To this end, our methods herein extend along an orthogonal direction and can be combined with existing architectural and algorithmic designs to obtain an optimized system as a whole.

3 A Multi-Layer Caching Model

Given the great diversity among applications of remote visualization, we need a unifying model of caching for remote visualization to systematically study optimization methods for cache configurations. A robust software infrastructure implementing the model is also required so actual experiments with different applications can be conducted under a consistent framework. Accordingly, we first describe our generalizing concept based on “incremental”, then the overall configuration of a cache in remote visualization. The general characteristics and optimization of the problem space are reserved for Section 4.

In Figure 1a, we illustrate a general setup of remote visualization applications. By choosing different types of connections between functional nodes, one could come to a number of possible setups. For example, a popular instance is shown in Figure 1b, a cache node is co-located with the client and the server, with a wide-area network separating them. The two cache nodes are referred to as Layer One (L_1) and Layer Two (L_2) caches.

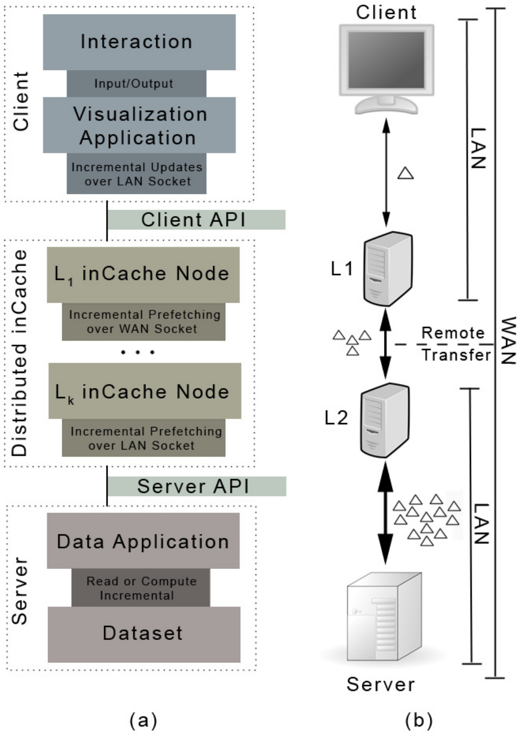


Fig. 1. A general model of caches based on the concept of *incremental*. (a) A conventional setup of remote visualization applications. (b) A common instance of how a remote visualization application is laid out.

3.1 Incremental

To generalize requested information over different applications of remote visualization, we define an *incremental* as the portion of data, be it intermediate or raw data, to be delivered across the network to the client upon request. For instance, in [2, 15], an incremental is composed of the texture and geometry that are constructed for each new view angle and depth range. Hence, it is both view-dependent and depth-dependent. In [12], a volume rendered image goes through wavelet compression before being transmitted across a dedicated ATM link. The client then receives the compressed image for each view as a new incremental update. When extracting view-dependent iso-surfaces, significant accelerations could be gained by focusing computing resources on extracting triangles that are appearing at each new view angle and/or new depth [9,13,14]. In this case, incrementals consist of those newly extracted triangles. Previous researchers have also proposed to package compressed data on a coarse level of granularity into larger groups, with each incremental corresponding to a set of similar views [6]. To summarize, applications of remote visualization, in general, pull incrementals through a chain of network nodes (k levels of cache), with each acting as a server.

In all cases, an incremental is identifiable by a *com-*

pound key of an arbitrary number of indices, each corresponding to a parameter in a user request, e.g. time step, view angle, iso-value, level-of-detail, etc. A compound key can then be encoded as a string by concatenating the name of the dataset with the list of indices. With keys represented as strings, we can use classic string methods to conveniently hash and manage incrementals of an arbitrary remote visualization application. Moreover, prefetching and deletion schemes can also be succinctly specified with efficient and general string operations.

With the concept of incremental, it is then possible to implement one common infrastructure to support caching needs of various remote visualization applications.

3.2 Cache Configuration

In our model, each cache node is independently controlled. Maximum size of a cache specifies the threshold before deletion must take place. On all cache nodes, deletion is performed using least recently used (LRU). The amount of incrementals removed with each deletion is controlled by an adjustable parameter.

Due to the large index space, we find it hard to organize incrementals using multi-way associative schemes like those in processor cache. Instead, we feed the key and incremental into a general hash function and maintain all incrementals in a hash table. Similarly, while processor caches usually prefetch in units of cache line, such a scheme is too simple to be used in remote visualization. We instead leverage the concept of neighborhood.

Suppose a dataset named *dn* is indexed by *m* variables, i.e. var_1, var_2, ..., var_m. The neighborhood, *S*, of an incremental *in* with a key, *Key*, can be specified as:

$$\begin{aligned} Key(in) &= dn.var_1.var_2.\ldots.var_m. \\ S(in) &= \{s : Key(s) = dn.var_1 \pm \Delta_1.var_2 \pm \Delta_2.\ldots.var_m \pm \Delta_m.\} \end{aligned}$$

The Δ's define the size of the neighborhood around incremental *in*. Since all prefetches are done in units of neighborhood, we refer to those Δ's as *prefetch increments (Pinc)*, with each Pinc controlling the range of an individual dimension (i.e. variable) of the dataset. By setting constrasting Pinc values for different dimensions, the flexibility to prefetch more aggressively in a certain dimension is offered. However, all incrementals in the same prefetch neighborhood are treated with the same priority.

Here we make a special note. In real-world scenarios, deletion and prefetch policies would likely need to be application specific. In this work, the deletion and prefetch policies are independent of the framework. As long as such policies can be procedurally described, our framework can invariably leverage them. Our current deletion policy based on LRU and prefetch policy based on neighborhoods are intuitive to define and use. The observed results (Section 6) seem reasonable with our selection.

4 The Problem Space

In an attempt to design an approach to cache optimization, we study average fetch times and the overall cache configuration space. In particular, we observe fetch times resulting from configurations randomly selected in the entire problem space. This is done to explore the possibility of an analytical approach for finding good cache configurations. We show that this is a virtual impossibility and discuss numerical minimization as an alternative.

4.1 The Target Function: $F(X_1,...,X_k)$

The specific aim of this work is to design a systematic approach to discover efficient (optimal or at least close to optimal) configurations of multi-layer caches in a remote visualization system.

Due to the dynamic nature of human interaction on the client-side, we can only take averages of fetch time, F, over a number of user requests and minimize that. The great variety of visualization applications also dictates such optimization be done in a case by case manner.

The input of F consists of X_js, with each X composed of the entire set of configuration parameters for one of the k caches. Hence, we have $F(X_1,...,X_k)$ as the target function, where the lower the function value the better. In addition, as network traffic varies during a period of time, e.g. different hours during a work day, the value of $F(X_1,...,X_k)$ would vary as well. The minimization process of $F(X_1,...,X_k)$ would then need to be re-done. Fortunately, as we will show in Section 6, this process completes with sub-second efficiency. Thus, although $F(X_1,...,X_k)$ is time-dependent, we do not need to include time as an input of F for the purpose of optimization.

4.2 Configuration Space

Supported by inCache (Section 5) and using the layout illustrated in Figure 1b, we tested three applications: dynamic streamings of (i) compressed images volume rendered for different view angles, (ii) iso-surfaces for a set of iso-values and time-steps, and (iii) streamlines for interactively specified seed locations (in both spatial and temporal coordinates).

Each level of cache, L_1 or L_2, is controlled by an independent set of configuration parameters: cache size defined in number of bytes, and sizes of deletion and prefetch (both defined in number of incrementals). Here we use the same prefetch size on all indices of the dataset, i.e. treating all indices with the same priority. Thus, instead of keeping multiple prefetch sizes, Δs, we just main-

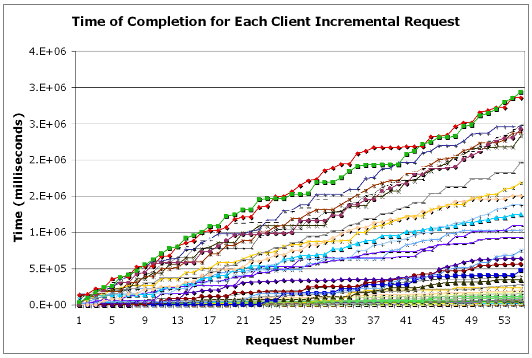


Fig. 2. Wall clock time plots of user requests under approximately 25% of the 200 configurations.

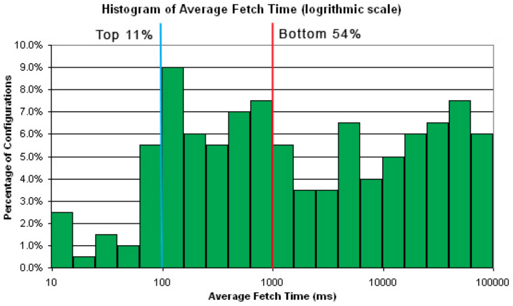


Fig. 3. Histogram distribution of average fetch time (in logarithmic scale), recorded from 200 random L1-L2 configurations of distributed caches, recorded when streaming a database of volume rendered images of a time-varying simulation dataset.

tain one universal Δ value. Therefore, with two levels of caches, L_1 and L_2, there are a total of 6 configuration parameters (3 for each cache). Note, however, the Pinc parameters are treated individually in actual optimizations.

The 6-dimensional configuration space is overwhelming to exhaustively search. Since we just need a qualitative sense of the overall configuration space, we start by randomly selecting 200 cache configurations, out of the 6-dimensional space, for each application. The cache sizes were distributed within the range from $2^{13} = 8K$ bytes to $2^{27} = 128M$ bytes, to match real-world scenarios. Deletions were chosen as a number of incrementals representing a random percentage of cache size. The Pinc values were picked as random numbers between 0 and 6, since anything larger would cause prefetching to become too costly. With each configuration, we run a scripted sequence of 60 random user requests. There is a one-second pause after a set number of requests have been answered to “mimic” a repeatable set of user interactions. The pause occurs after 3, 1 and 30 requests for applications (i), (ii) and (iii), respectively.

Following an initial start-up cost, efficiency with the subsequent requests should reflect caching performance. Using the recorded times of the last 55 requests, we analyze: (i) the effects of cache misses on the wall-clock time, and the distributions of (ii) average fetch time and (iii) “good” configurations in the configuration space. Interestingly, all three applications produced similar results. The results from application (i) follow.

In Figure 2, we plot the wall-clock times when each of the last 55 requests was fulfilled. Each curve in Figure 2 corresponds to a different configuration. To remove visual cluttering, we show only approximately 25% of all curves. Cache hits and misses are apparent from observing the stair-stepping effects in the figure, with each miss showing a noticeable shift in the graph. Lower overall slope corresponds to better configurations. This qualitative characteristic is present with all three applications.

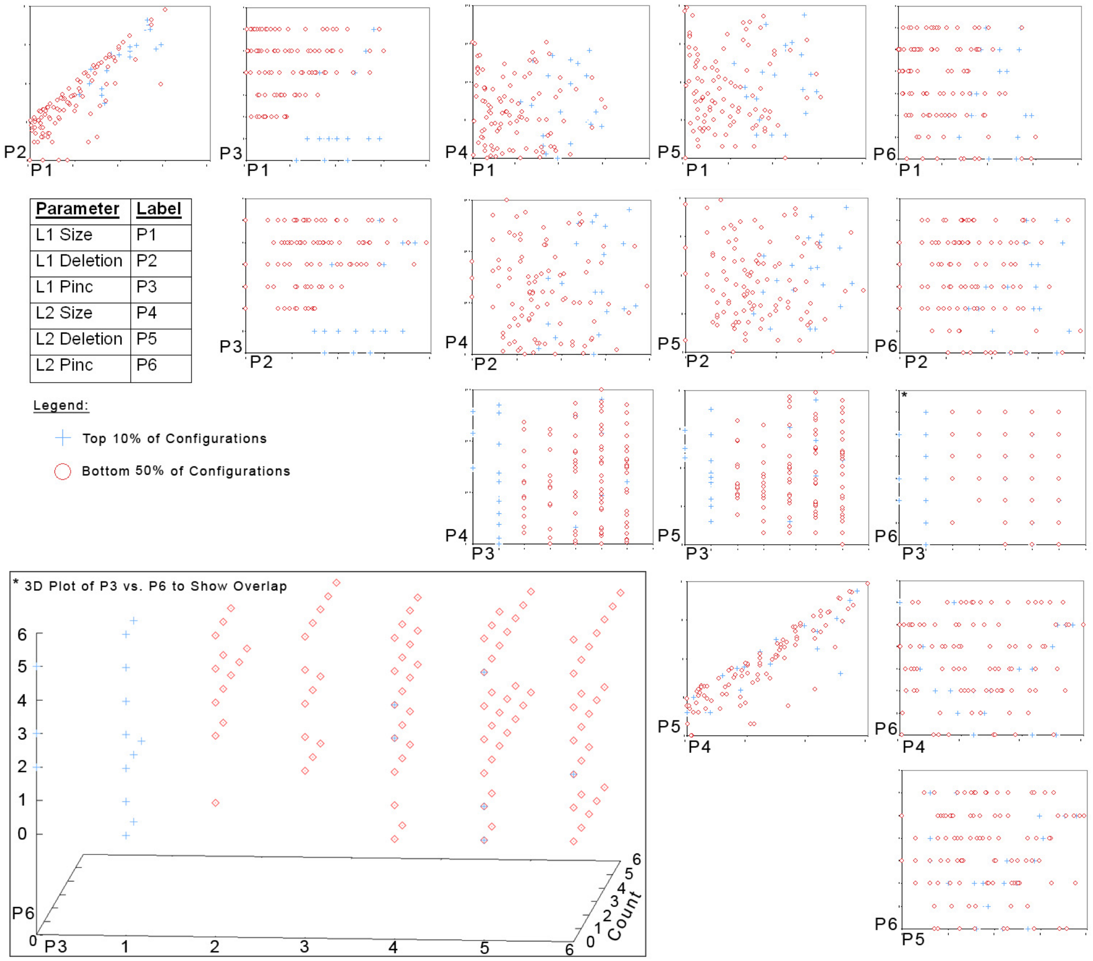
We also construct a histogram of average fetch time to show the corresponding distribution. The histogram shown in Figure 3 has the horizontal axis, the time, organized in logarithmic scale. In other words, the bins are exponentially sized. As shown, the average fetch times span across four orders of magnitude, with the first containing the best 11%.

After seeing histograms similar to the one in Figure 3 for each application, we are convinced that finding a good (not even optimal) configuration is significant. A random cache configuration, or one manually chosen or “twiddled” will not guarantee efficiency, whereas a systematic way of searching for such a configuration is desirable.

To design a proper search method, a qualitative understanding of the overall problem space would be very helpful. We developed the x-y plots of the top 10% vs. bottom 50% of configurations in Figure 4 to show, for instance, the correlation between the sizes of L_1 vs. L_2 caches.

Some well-known patterns can be observed, such as L_1 cache should not prefetch more than L_2 cache. L_1 and L_2 should not be approximately the same size, and either needs to be large but not both. Aside from those, there appears to be little commonalities present in all “good” configurations, which are also highly dispersed throughout the entire configuration space.

To rule out the possibility of some higher-order relationship among the parameters in“good” configurations, we resorted to Principal Component Analysis (PCA) as well as non-linear decision-tree classification. With PCA-analysis, we found four out of six significant eigenvectors in the eigen-space, i.e. the 6-dimensional space can



Parameter	Label
L1 Size	P1
L1 Deletion	P2
L1 Pinc	P3
L2 Size	P4
L2 Deletion	P5
L2 Pinc	P6

Fig. 4. A qualitative view of configuration parameter space distribution, using x-y plots of top 10% vs. bottom 50% configurations, in the application that dynamically streams compressed volume rendered images.

be reduced to a 4-dimensional space, preserving 90% of variability among the data points. While this reduction is helpful, the actual problem space is even larger due to multiple Δs to handle. Hence the dimension reduction still generates a rather complicated space. With decision-tree classification, we resorted to the mature body of work [19] used in data mining. The precision of determining whether a configuration is good consistently ranks below 70%, with a sensitivity as low as 39%. In short, we could not find any reliable structures in the space of configuration parameters distinguishing good from bad.

4.3 Numerical Minimization

Not knowing the analytical form of $F(X_1,...,X_k)$, we cannot solve for extremal points in the N-dimensional domain. Given the immersed nature of good and bad solutions (Figure 4), we decided to use numerical minimization as a heuristic to iteratively search for a better setting.

By abstracting all cache parameters from all levels of cache as an N-dimensional vector, our target function $F(X_1,...,X_k)$ becomes $f : R^N \rightarrow R$. Supposing n is the total number of parameters to configure a cache, and k is the number of levels of cache, then N is equal to $n \times k$. We used two alternative procedural numerical minimization algorithms: the gradient based (i) Steepest Descent and (ii) Conjugate Gradient.

While Steepest Descent is the most straightforward method of numerical minimization, Conjugate Gradient is more theoretically sophisticated and may yield better results. As we will show in Section 6, between the two methods, one does not always outperform the other. It is of particular value to use both methods at run time and return the best configuration for more robustness.

We implemented both methods, using finite differences to approximate gradient. Specifically, we modify one of the N parameters while fixing all the other $N-1$ parameters. Since there are two possible directions of variation with each of the N parameters, a reliable estimation of gradient requires $2N$ measurements of fetch time. Such an $O(N)$ complexity caused experiments to take hours to complete, and prompted a need for simulation-based optimization. A calculation of the Hessian of this function would cost $O(N^2)$ measurements of fetch time, rendering Newton's Method and many of its variations prohibitively expensive for this problem.

The direction of Steepest Descent is simply the negative of the gradient. Conjugate Gradient calculates a different direction, based on the gradient, that in practice tends to more accurately reflect the actual minimum. The first iteration of Conjugate Gradient is equivalent to Steepest Descent. Typically for Conjugate Gradient, a step size search is required, but because of the inherent discrete nature of our problem, it is not needed for our calculations. Also, many linesearch algorithms require additional gradient calculations; even the simplest of these at least requires extra function evaluations [5] . Additionally, Conjugate Gradient is known to sometimes give bad directions, so an occasional restart is needed. We implemented the Polak-Ribiere version of Conjugate Gradient [18], which has an elegant fix of this shortcoming.

The optimization (numerical minimization) is performed independently when an application is started. We begin with an initial configuration and measure fetch times either through actual experiments (Section 5.3.1) or via simulation (Section 5.3.2). We then use these measurements to determine the direction of minimization.

Note that we use numerical minimization merely as a heuristic search for a better solution. Ideally we would prefer to find a global minimum, but doing so is generally difficult and time consuming. Incoming user requests, network instability, and variable sized blocks of data are inherent in all remote visualization systems. Consequently, there is a small likelihood of a globally optimal solution remaining optimal for long enough to justify finding one. Therefore, we opt for finding any better solution in a short amount of time, i.e. within one second.

The properties of f, that numerical optimization routines require, are used to bound approximations and guarantee convergence. However, our criterion for convergence is simply whether the next configuration yields a slower fetch time. We continue as long as the fetch time is faster than the previous iteration. Indeed, we find that the direction returned by either Steepest Descent or Conjugate Gradient, and a move from one cache configuration to another in that direction generally reduces our average fetch time. Finally, we recognize that we have hardly tried all possible methods of numerical minimization. More sophisticated algorithms could potentially lead to better numerical performance. As we will show in the Results section, our overall framework is already effective without utilizing more complicated methods.

5 THE INCACHE SYSTEM AND OPTIMIZATION

5.1 The inCache Infrastructure

5.1.1 Design Concepts

While designing inCache, we hoped to develop a sufficient infrastructure for investigating the problem of multi-level cache optimization, and at the same time have the resulting package be of value for practical use.

Each cache would hold all incrementals in a contiguous repository, indexed through a hash table. In the meantime, a separate priority queue is used to maintain the LRU order of all incrementals. Accordingly, the priority queue is keyed on the last time each incremental is accessed. All reads/writes in the hash table take $O(1)$ time, while inserting/deleting an incremental from the heap takes $O(logM)$, where M is the total number of incrementals in a cache.

Another issue that inCache handles is the need to frequently change how a cache is configured. With each inCache being an independent process running on a networked node, there are three interfaces to reconfigure inCache: (i) manual input through the inCache process' console prompt, (ii) periodic update from a textual configuration file on local disk (a mutable option), or (iii) dynamic alteration through a socket. When an inCache process is directed to reduce its size, it deletes present incrementals in LRU order until the request is met. However, when a larger size is given, the inCache process will cap that value by the size of its main memory.

5.1.2 The inCache Nodes

Our overall architecture follows the illustration of Figure 1a. The inCache package solely provides the functionality of multi-level caching. Server and client are the responsibility of application developers.

Every inCache node runs an independent process, acting both as a producer and a consumer. (i) As a consumer, it takes in data by prefetching neighborhoods of incrementals from a lower-level cache or directly from the server. Since all incrementals are identified by string-type keys, an inCache process makes no distinction among datasets or applications. (ii) As a producer, an inCache process answers requests from higher level caches or the client. If it does not hold the requested incremental, i.e. a cache miss, it makes a prefetch request to a lower-level cache, and waits for the prefetch to arrive before answering the original request. This chain of requests ends at the remote server, where all requests result in a hit. As a request travels from a client to a remote server, each level of cache usually receives and holds a larger subset of the data, illustrated in Figure 1b with an increasing number of triangles beside each inCache node.

Table 1. The inCache Interface.

Client API
connect_cache(hostname, port, type)
client_retrieve(dataset_name, indices[], bufferout, cachefd)
client_reconfig(config_lines[], cachefd)
Server API
connect_cache(hostname, port, type)
get_lookupinfo(dataset_name, indices[], increments[], cachefd)
server_insert(dataset_name, indices[], data_item, cachefd)
server_invalid_request(dataset_name, indices[], cachefd)

All connections to inCache processes are through sockets. If the targeted inCache process actually runs on the same host as the consumer process, then the communication goes through a very low overhead socket loopback. When that is not the case, data transmission is then in TCP/IP. The user can benefit from this design by choosing the best node layout for a specific network, chaining additional nodes until the final data server is connected.

A final consideration is to transmit requests and data as quickly as possible, so separate threads within each inCache process are used to handle incoming incremental requests from its client and incoming incremental data from its server. Thread control and mutex locks became the most demanding task to achieve robustness as well as efficiency during inCache development. However, using threads enabled us to handle multiple clients per node, and hence, several different users may simultaneously access the inCache system. In this respect, every inCache node works very much like a web server.

5.1.3 The External Interface

While the intrinsics of the inCache infrastructure are rather involved, it is sufficient for application developers or users to deal solely with a compact external interface, as shown in Table 1, where the functions are grouped into client API and server API. The locations of the two APIs in the software architecture is also illustrated in Figure 1a. The underlying operations within an entire group of inCache nodes, including prefetch and cache deletion, etc., are transparent to users.

To chain inCache nodes, a client can connect to its producer cache node through a given socket's address by the connect_cache() function. With the established connection, a client makes an incremental request with client_retrieve() and waits for the data to be returned. The producer inCache node will then immediately reply with the data on a hit, or translate the request into a prefetch and send it to the next linked node on a miss.

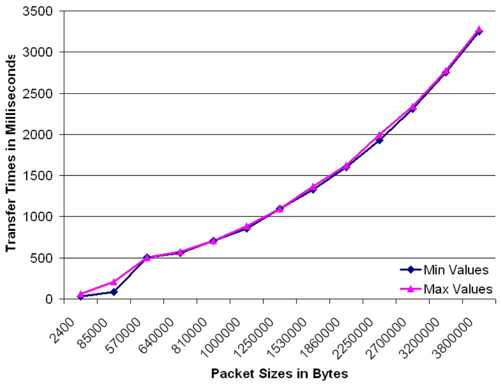


Fig. 5. WAN bandwidth measured for different message sizes.

The lower node could be another cache or the actual data server. The server will respond to this request by retrieving the incremental lookup information via get_lookupinfo() and then determining the availability of the data. Once the data is ready, the server can insert the information back into the inCache node with server_insert(), or if the lookup returns no match, the server will respond with an invalid incremental message with server_invalid_request(). The client_reconfig() function allows the consumer client to configure any number of layers by specifying the configurations in a chain of strings that propagate down.

5.2 Testing Framework

To comprehensively study the performance aspects of inCache, we have designed the following testing framework. We have tested inCache with multiple applications under a variety of network environments.

5.2.1 Network Layout

For our model (Figure 1a), remote streaming using a client, a server, and L_1 and L_2 caches, is the most practical. The standard components of any remote system, a server and a client, represent the end layers. The server acts as the compute layer responsible for the storage of and the access to pre-computed visualization data. The middle layers consist of L_1 and L_2 caches joining the server and client. In all tests, we use the standard Internet as the wide-area network. Figure 5 illustrates the typical bandwidths we obtained for varying network message sizes. Noting that our framework is independent of the type of network used, better system performance could be possible if quality-of-service is employed.

A cluster located in North Carolina State University (NCSU) was used for the server machine, which transfered data to the client located at the University of Tennessee. Experimental tests were also conducted with the remote server running on a workstation at the Ohio State University (OSU), but due to having nearly identical network bandwidth, the tests results were very similar. Hence, herein we only show results obtained using the NCSU cluster as the server.

In total, we evaluated 4 different network layouts. In the first two layouts, the client resides on a 100Mbps Ethernet LAN, whereas in the last two layouts the client is accessed via Internet over an 11Mbps wireless LAN. In all cases, the WAN separates the client from the server.

- Layout 1 has L_1 on the same ethernet LAN as the client and L_2 on the same LAN as the server, but not on the same machines. L_1 and L_2 handle all network transmission on the WAN connection.
- Layout 2 places both L_1 and L_2 caches on the client side (i.e. no cache on the server side). L_1 is on the same machine as the client. L_2 cache on the same ethernet LAN as the client. L_2 and the server handle all communication across the WAN. This layout was shown to be useful in [6].
- Layout 3 is similar to Layout 1, with a wireless network replacing the ethernet LAN on client side. The WAN connection remains unchanged.
- Layout 4 is similar to Layout 2, with a wireless network replacing the ethernet LAN on client side. The WAN connection remains unchanged.

5.2.2 Modeling User Requests

Suppose there are m indices in a dataset, with each index in the compound key being Key_j. We assume that a user would modify any given compound key in consecutive steps by issuing a new request with each modification. We further assume an average user will take one step of unit size in a randomly chosen direction, j. Accordingly, our random step generator begins with an initial Key and randomly picks one of the m indices, Key_j to increment or decrement (with an equal probability) by a unit step.

Using such a procedural user model, we can conveniently control the user behavior by setting a different rate of user input. Between experiments we can use the same random sequence of user requests to measure differences in fetch time in a controlled manner.

If a user is more likely to change a parameter than other parameters, we can model this behavior by setting different probabilities for each Key_j to be chosen for next move. As an example, if a user engages in browsing a hierarchically organized data at a low level-of-detail (LoD), user requests can be generated with a fixed LoD index, while all other indices are freely variable.

The inCache system treats all requests the same. Over a sequence of user inputs, it is for the minimization code to discover that, for example, an index like time should incur larger prefetch (high Pinc value) or an index such as LoD should have a Pinc value of 0. In this way specific application requirements are implicitly handled.

5.2.3 Testing Applications

In our tests, the visualization results to be streamed across the network have been pre-computed. This is reasonable for the scope of this paper, because the time to compute an incremental on the fly could be treated as a part of the general network latency. The same inCache model still applies. Our tests include four applications, chosen to reflect incrementals of constant vs. varying and small vs. large sizes, and different types of compound keys.

- Application 1 streams a set of streamlines of equal length, computed from the well-known tornado dataset. There are 46^3 separate streamlines, with each being 2.4 KB in size, indexed by the coordinate of the seed of each streamline. The client request rate is 30 requests/second.
- Application 2 involves a database of volume rendered images of a 30 time-step simulation data. The images are 512×512 in resolution, indexed by time step and two spherical coordinates (72×60) describing the view angle. With Z-lib compression, image sizes average 85KB, ranging from 70KB to 110KB. The simulated client makes 3 requests/second.
- Applications 3 and 4 deal with a set of iso-surfaces extracted at 11 different iso-values, from a 99 time-step volume dataset. After lossless compression using Z-lib, the sizes of the compressed meshes vary widely between 0.5MB and 3.8MB. Application 3 and 4 are tested at 1 request/second and 3 requests/second, respectively. User varies the time index over iso-value with a 2-to-1 probability.

5.3 Optimizing inCache

5.3.1 Optimizing Fetch Time by Experiments

By experimenting with real-world applications in real-world scenarios, we tried to verify the effectiveness of numerical minimization. An initial cache configuration is chosen when each application is started. We then rely on either Steepest Descent or Conjugate Gradient to determine how to vary the configuration in the next iteration.

We note here that different types of cache configuration parameters may have drastically different scales. For instance, cache size could range from Kilobytes to Gigabytes, while Pinc (size of prefetch) of each index used by the dataset may only be of small values like 1, 2, 3, etc. Hence, when Steepest Descent and Conjugate Gradient try to evaluate various gradients, we have to use step sizes empirically determined for each parameter in consideration. For instance, the step size for "cache size" is always 10% of the previous size. Deletion size is given in terms of number of incrementals. The step size of deletion is also a percentage of the maximum number of incrementals the current configuration could contain.

While deciding the initial configuration, we do have one condition to always meet. That is the initial cache size must be large enough to hold at least one half of the entire prefetch neighborhood for that cache. We also do not allow Pinc values to go beyond 6. Typically, remote visualization datasets are indexed by multiple (more than 3) indices, and with Pinc $= 6$ the neighborhood for prefetch is already exponentially sized.

As we will show in Section 6, both methods of numerical minimization worked well in finding good cache configurations. However, their shortcomings are also significant. In either, before taking the next step, one needs to make a large number of trial configurations, each with an orthogonal change from the current one. Under a configuration, to obtain a reliable measure of average fetch time, we run through a randomly generated sequence of user movements. Even when only dealing with 10 parameters ($N = 10$), for instance, Steepest Descent may require hours to just take one step in experiment. An optimization experiment could take more than a day to finish.

To resolve this bottleneck, we have developed a highly efficient simulation based approach to carry out the optimization. This approach allows an optimization to be computed within a second of time using just one CPU.

5.3.2 Optimizing Fetch Time by Simulation

To address the experimental bottleneck, we provide a simulation of the inCache infrastructure that calculates fetch times based on average network measurements.

We start by collecting transfer times, or bandwidth, over the wide-area network for a range of message sizes. This test is run on every link on our targeted network connection. The measured network bandwidths can then be plotted as a function of message size for each link. We found that the wide-area network demonstrates stable performance over a reasonable time span.

When a configuration is chosen, the same random sequence of user movements is considered, with the overall operation of the inCache system simulated. The fetch time of each network message, however, is obtained simply by linear interpolation according to message sizes from the model obtained above. The final fetch times used by numerical minimization are still computed as an average over the entire sequence of user movements. This way, the most time consuming step during the optimization process is replaced by efficient computations.

The shortcoming of our approach stems from the difficulty of handling fluctuating networks and quickly changing rates of user movements. Fortunately, because of the speed of the simulation, it is reasonable to rerun the optimization at any time and reconfigure the cache configurations at run-time. Also, the simulation takes as input the estimated amount of time between user requests, which also allows for a rerun and reconfigure. The results of this simulation (provided in Section 6) are quite similar to those of the experiments.

Table 2. Request Rates by the Client per Second.

	App 1	App 2	App 3	App 4
Avg Inc. Size	2.4 KB	85 KB	1.8 MB	1.8 MB
Req. Rates	30/s	3/s	1/s	3/s
Requested Data	72KB/s	255KB/s	1.8MB/s	5.4MB/s

6 Results and Discussions

6.1 Evaluation of Optimized Performance

Since our main approach is to minimize average fetch time, let us first evaluate the resulting performance after optimization. In Table 2, we list the request rates and average fetch sizes for Applications 1 through 4. It is important to note that different settings of prefetch would cause a different amount of network traffic between L_1 and L_2, and L_2 and the server. Here we are solely concerned with the link between L_1 cache and the client, where there is always only one incremental returned for each request.

As described in Sections 5.3.1 and 5.3.2, there is not a clear winner between the two alternative methods of numerical minimization, Steepest Descent and Conjugate Gradient. However, since the simulation consistently completes with sub-second efficiency, we can always afford to run both methods and choose the better convergence result. We then set up an actual experiment using the resulting configuration and record the average fetch time by running through the scripted sequence of user movements. From that, we compute the rate at which requests are fulfilled. Note that this shows the performance perceived by the client. Several interesting features can be observed in Figure 6, a graphical plot of this result.

First, intuitively, the more data requested per second by the client, the slower the requests can be answered. The

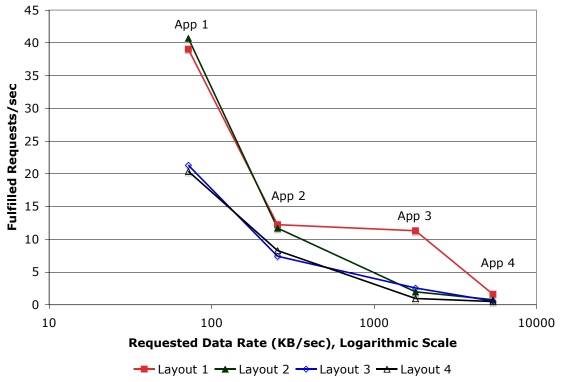


Fig. 6. Evaluation of performance after optimization. The horizontal axis is the rate at which the client requests data (in KB/second), while the vertical axis shows the corresponding actual rates at which the system can fulfill client requests (in number of requests/second). On each curve (for each layout), from left to right, are Applications 1 through 4.

curves essentially depict the lower bound of achievable latencies for each network layout with the given combination of fetch size and user request rate, i.e. the lower bounds are specific to the pattern of data movement. From a lower bound, we can tell whether an algorithm is appropriate in a scenario. In the scenarios in Figure 6, Layout 1 likely supports interactivity for all four test applications.

Second, Layout 1 and 2 performed considerably better when the rate of data requests are on the order of several hundred KBs/second or lower. But when rate of data requests go up to 1.8 MB/second, only Layout 1 maintained better performance. This difference among layouts almost completely diminishes when data is requested at 5.4MB/second. We can thus deduce that having two caching nodes residing on opposite ends of the slowest link of the network would be most helpful to remove the performance bottleneck. For instance, in Layout 1, the slowest link is the WAN between L_1 and L_2 caches, while the LAN connections on either the server or the client side are not the bottleneck. Of course, when the data requests go up to 5.4MB/second, even the LAN connections start to get congested.

For the applications tested, the wireless network only delivered about half of the performance that LAN environments did. This is probably because wireless networks are more easily saturated. But after data request increases so much that all links become saturated, the impacts caused by differences among ethernet LAN, wireless and WAN are then minor.

Layout 3 and Layout 4 are also different, yet they have similar results in nearly every application. With Layout 3, the cost of using the slower wireless network must be paid for every request, including hits, since the L_1 cache is located across the LAN. However, Layout 4 has L_1 directly on the client, with near instant hits on L_1 but a higher cost of misses and network transfer for prefetch requests made by L_1. As discovered by our optimizer, neither trade-off seems to provide a better layout since the lower bounds of fetch time are nearly the same.

On the one hand, an interactive application must at least maintain 10-15 frames/second according to current standards. In this context, we find Layout 1 and 2 sufficient for all 4 applications to be interactive as long as the client needs less than 500 KB/second of data. On the other hand, some ingenious remote visualization algorithms might only need a new update every 2 seconds, for instance. Then, with sufficient caching, even the wireless network supports interactive remote visualization, as long as data requests are slower than 5.4MB/second. Note, in Figure 6, even the lowest data point corresponds to an ability to fulfill 1 request every 2 seconds. From similar analyses, a guidance of which network layout to choose or what algorithm to use can be obtained.

The applications we tested represent a generic spectrum of common visualization applications. To set up the testing framework, very little work was needed. We finished developing each application within a couple of days due to the simplistic API of the inCache infrastructure. It should be relatively easy for other mature visualization applications to adopt our approach.

We realize it is hard to prove that our approach actually finds the optimum. However, since the performance results are quite close to what a dynamic application can obtain from today's network infrastructure, we deem the resulting performance to be close to optimal.

6.2 General Observations

In total, because of the two minimization methods, both in experiments and simulation, and in all the 16 combinations of network layout and testing applications, we already have 64 different scenarios without counting additional test runs to cross-check experiments versus simulation. Due to space limits, we cannot include detailed configuration results of the optimization process for all the tests. We have compiled that information as supplemental material to this paper.

In this section, we discuss some general findings that we made by analyzing the cache configurations resulting from optimization. These findings are unexpected, and may be specific to various scenarios (i.e. the combination of application and system settings). We present these findings to demonstrate the advantages of runtime optimiza-

Table 3. Some statistics of cache sizes (MB)

	App 1	App 2	App 3	App 4
Min	0.031	1.7	51.5	7.1
Max	0.45	15.0	234.9	214.7
Average	0.28	5.4	129.9	55.4

tion from perspectives different from those in Section 6.1.

Let us also briefly describe typical parameter values observed during optimization as a basis of reference for phrases like “small prefetch” or “mid-range cache size”. Our applications are data intensive. It is quite common for at least one of the Pinc dimensions to be zeroed out during optimization. We consider a small prefetch to be *Pinc* < 2. *Pinc* > 4 almost always leads to inferior fetch times. In Table 3, we collected the minimum, maximum, and average cache sizes (without distinguishing L_1 and L_2), after optimization via experiment and simulation, for each application. Obviously, mid-range cache size is application dependent by nature.

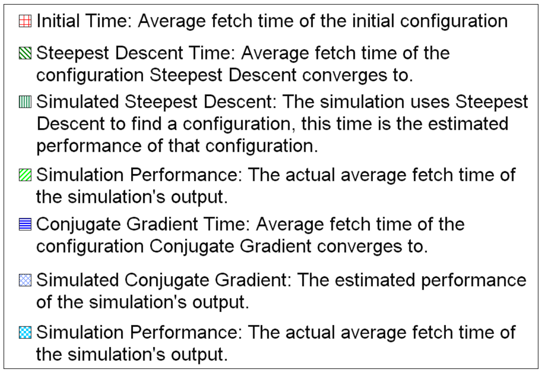
6.2.1 Impacts of User Interaction Rate

User request rate is probably more important than previously anticipated. Due considerations must be given to the targeted user request rates to design algorithms and systems with real world applicability.

If, when prefetch is costly, the user does not leave sufficient time between requests to allow for prefetch, prefetching could quickly become a burden. For instance, in Application 1, at 30 requests/second the optimized cache configurations use minimal cache prefetching, even when each incremental is of a mere 2.4 KB. With Application 4, a lower request rate (3 requests/second) is used, and each of its incrementals takes about 0.1 seconds to transfer across the WAN. A medium Pinc value of four for each of the two indices would cause $(2 \times 4 + 1)^2 = 81$ incrementals to be prefetched together. Then, prefetch becomes a wasteful “post-fetch”.

Similarly, unless slow rates of data requests are used, the system benefits less from prefetching across the WAN than across a LAN (i.e. high vs. low prefetch costs). Often the L_2 cache, when close to the server node, can take advantage of some prefetching to improve L_1 misses. Such behavior is consistently reproducible in both experiment and simulation. This finding justifies the superiority of latency hiding algorithms. By separating user interaction rates from rates of data requests, a slower rate of data requests is possible and leads to better performance.

Applications 3 and 4 serve as latency hiding examples. When a surface mesh is requested at 3 requests/second (Application 4), high fetch times are shown to occur. Since mesh rendering is fully interactive by itself, we slow to 1 request/second in Application 3. Fetch times were reduced dramatically because the cache was able to utilize the “idle time” for more prefetch on both L_1 and L_2. In contrast, the optimizer used very small prefetch in Application 4 to compensate for the higher request rate.



⊞ Initial Time: Average fetch time of the initial configuration

▧ Steepest Descent Time: Average fetch time of the configuration Steepest Descent converges to.

▥ Simulated Steepest Descent: The simulation uses Steepest Descent to find a configuration, this time is the estimated performance of that configuration.

▨ Simulation Performance: The actual average fetch time of the simulation's output.

▤ Conjugate Gradient Time: Average fetch time of the configuration Conjugate Gradient converges to.

▩ Simulated Conjugate Gradient: The estimated performance of the simulation's output.

▦ Simulation Performance: The actual average fetch time of the simulation's output.

Fig. 7. The legend for Figures 8 and 9.

6.2.2 Impacts of Network Speeds and Incremental Sizes

Network speeds have the expected effect on fetch times, i.e. the lower the network bandwidth the longer the fetch times. Similarly, larger incremental sizes result in longer fetch times. For datasets with uniform incremental sizes, cache optimizations are more consistent and predictable. However, we discovered with Applications 3 and 4, whose incremental (isosurface) sizes vary significantly for different iso-values, that fetch times can widely vary depending on which incremental is under request. A small change in *Key* value can lead to a substantial jump in resulting fetch time. This makes optimization difficult. One way to overcome the problem is to continuously optimize inCache as the user interacts with the visualization. This is why we have used the sequence based user model.

Also, a portable client placed on a wireless device benefits from the inCache system. The greatest increase in performance occurred when the clients were able to access a large portion of data on a local network, i.e. when both L_1 and L_2 cache nodes appeared on the same LAN as the client. L_2 acted as a larger storage unit, allowing even L_1 cache misses to incur a small penalty.

6.3 Experiments vs. Simulation

Here we present plots of the resulting average fetch time, measured in different cases, to demonstrate the intrinsics of the optimization process. Together, these plots are a summary of the supplemental material of this paper.

Each graph in Figures 8 and 9 represents one Application over the 4 different layouts (Layout 1-4). For each,

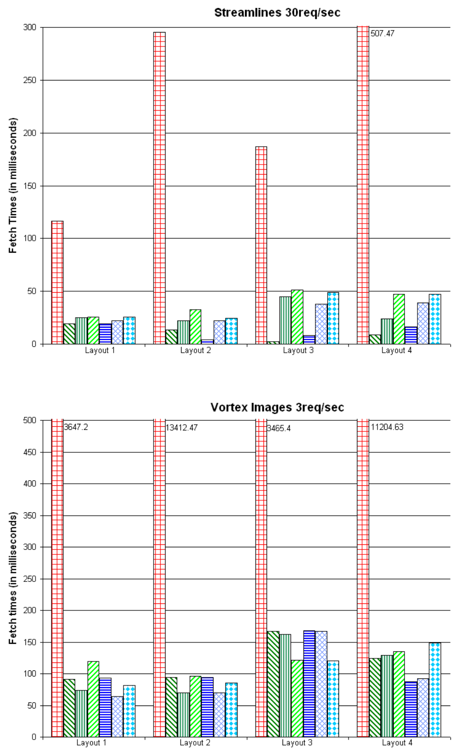


Fig. 8. Latencies measured in Applications 1 (top) and 2 (bottom).

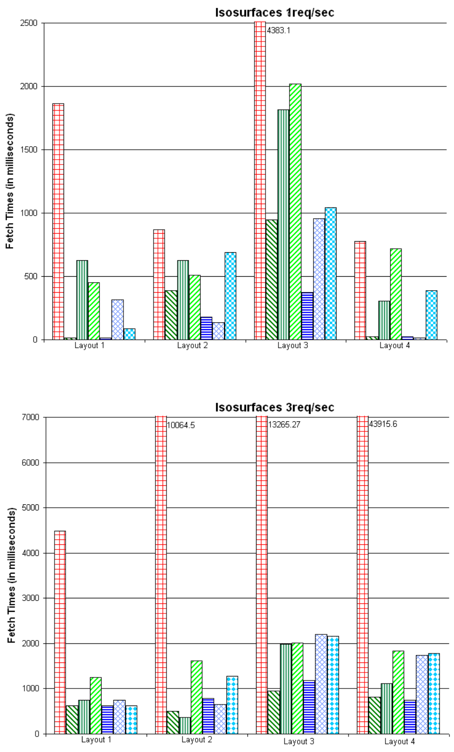


Fig. 9. Latencies measured in Applications 3 (top) and 4 (bottom).

we compare two optimization methods and experiment vs. simulation. For each application the same initial cache configuration is used.

For some tests, there is such a large range of initial fetch times and resulting optimized times that the maximum fetch time value allowed for the Y axis had to be capped to produce sensible plots. In that case, all values above the maximum range of the graph are shown in print on the graphs. With every combination of testing application and network layout, results from both the actual experiments and different methods of simulation are shown. Specifically, the legend is organized as Figure 7, with the graphs shown in Figures 8 and 9.

Our simulation calculates fetch times based on measured network bandwidth, greatly accelerating the process. For instance, both an hour long streamline experiment and a two hour long vortex experiment can be simulated in seconds. A four hour long isosurface test is simulated even faster because there are only two Pinc dimensions for isosurfaces, even though isosurfaces are much larger and more expensive to actually transfer. In all tests, simulation and experiment results are compared.

Although we have tested and found high stability on all the non-wireless networks we use (WAN and LAN), our simulation runs are likely biased. Nevertheless, as seen in the four graphs of all runs, even though the optimized configurations from simulation and experiments differ, the resulting fetch times are similar. However, the similarity is greatly affected by network instability and varying incremental sizes, as in Application 3. We ran the inCache system under all of the suggested outputs and the resulting fetch times were near the estimates, and, therefore, near the fetch times of the experimental runs. Unfortunately, the effect of network-related noise becomes quite apparent whenever a network layout includes a wireless network, which is notoriously sporadic. With the wireless network, our optimizer was still able to obtain a performance corresponding to roughly half of that delivered by non-wireless network.

6.4 Sensitivity to Initial Configuration

Due to space limits, we cannot provide in this paper a comprehensive sensitivity study of all the 64 testing scenarios in Figures 8 and 9. However, we believe our framework is reasonably insensitive to how initial configurations are seeded.

First, we chose the worst 10 configurations from the same set of 200 configurations tested in Section 4.2. The latencies recorded in those 10 configurations range from 36 to 53 seconds/request. All 10 configurations typically employ very large prefetch sizes. We optimize those configurations by simulation, using both Steepest Descent and Conjugate Gradient as the minimizer. By choosing the better result from the two minimizers, the optimized latency falls in the range of $[0.8, 4.2]$ seconds/request, with the mean being 2.6 seconds/request. Although we did not achieve as good of a result as in Section 6.1, we still obtained substantial improvements.

Second, for tests in Figures 8 and 9, each application starts from the same initial configuration, and in all cases consistent performance gains have been obtained after convergence. Since too much prefetch may very well be harmful, it may be practical to always start from initial configurations like the ones we used, where a small prefetch (e.g. $Pinc = 1$) is coupled with a mid-range cache size. Another practical strategy is to leverage the efficiency of our simulation. One can always start from a handful of reasonable initial conditions and choose the best among all the converged results, like how quicksort uses median-of-three to choose pivot.

7 Conclusion and Future Work

In this work, we focused on optimizing caching and prefetching in remote visualization systems. By way of real-world experiment and computer simulation, we have developed an approach, as well as an infrastructure, to seek close to optimal cache configurations maximizing network utilization. We envision several ways that our work could be leveraged. A developer can gauge whether a certain user requirement is realistic by the use of our simulation modules. An administrator can use our approach to determine more optimized layout for a system, and also dynamically re-configure cache nodes on the fly to maintain high performance.

In the future, we would like to develop a more comprehensive sensitivity analysis in a separate work. In addition, we intend to evaluate other optimization techniques. We also hope to explore using real user movement sequences captured from actual inputs by human subjects to better validate our approach.

Acknowledgments

We are thankful to our anonymous reviewers for their insightful suggestions on how to improve the paper. This work is partially supported by NSF grant CNS-0437508, the Scientific Data Management Center (http://sdmcenter.lbl.gov), and the Institute for Ultra-Scale Visualization (http://www.iusv.org) under the Department of Energy's Scientific Discovery through Advanced Computing program (http://www.scidac.org).

References

[1] W. Bethel, B. Tierney, J. Lee, D. Gunter, and S. Lau. Using high-speed wans and network data caches to enable remote and distributed visualization. In *Proceedings of Supercomputing*, Dallas, TX, 2000.

[2] Wes Bethel. Visualization dot com. *IEEE Computer Graphics and Applications*, 20(3):17–20, 2000.

[3] K. Brodlie, D. Duce, J. Gallop, Sagar M., Walton J., and J. Wood. Visualization in grid computing environments. In *Proceedings of IEEE Visualization*, pages 155–162, Austin, TX, 2004.

[4] M. Charney and T. Puzak. Prefetching and memory system behavior of the spec95 benchmark suite. *IBM Journal of Research and Development*, 41(3):265–286, 1997.

[5] J. E. Dennis and Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Society for Industrial & Applied Mathematics, 1987.

[6] J. Ding, J. Huang, M. Beck, S. Liu, T. Moore, and S. Soltesz. Remote visualization by browsing image based databases with logistical networking. In *Proceedings of Supercomputing*, Pheonix, AZ, 2003.

[7] K. Engel, O. Sommer, C. Ernst, and T. Ertl. Remote 3d visualization using image-streaming techniques. In *ISIMADE'99*, pages 91–96, 1999.

[8] K. Engel, O. Sommer, and T. Ertl. An interactive hardware accelerated remote 3d-visualization framework. In *Proceedings of Data Visualisation*, pages 167–177, Amsterdam, Netherlands, 2000.

[9] J. Gao and H.-W. Shen. Parallel view-dependent isosurface extraction using multi-pass occlusion culling. In *2001 IEEE Symposium in Parallel and Large Data Visualization and Graphics*, San Diego, CA, 2001.

[10] J. Gee, M. Hill, D. Pnevmatikatos, and M. Smith. Cache performance of the spec92 benchmark suite. *IEEE Micro*, 13(4):17–27, 1993.

[11] Pietrzykowski J. Decision support tool for web cache management. *Journal of Telecommunications and Information Technology*, (3), 2002.

[12] P. Li, S. Whitman, R. Mendoza, and J. Tsiao. Prefix – a parallel splatting volume rendering system for distributed visualization. In *Proceedings of Parallel Rendering Symposium*, pages 7–14, 1997.

[13] Z. Liu, A. Finkelstein, and K. Li. Progressive view-dependent isosurface propagation. In *Proceedings of Data Visualization*, Ascona, Switzerland, 2001.

[14] Y. Livnat and C. Hansen. View dependent isosurface extraction. In *IEEE Visualization '98*, pages 175–180, Research Triangle Park, NC, 1998.

[15] E.J. Luke and C.D. Hansen. Semotus visum: a flexible remote visualization framework. In *Proceedings of IEEE Visualization*, pages 61–68, Boston, MA, 2002.

[16] K.-L. Ma and D. M. Camp. High performance visualization of time-varying volume data over a wide-area network. In *Proceedings of Supercomputing*, Dallas, TX, 2000.

[17] A. Neeman, P. Sulatycke, and K. Ghose. Fast remote isosurface visualization with chessboarding. In *Proceedings of Parallel Graphics and Visualization*, pages 75–82, Austin, TX, 2004.

[18] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[19] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[20] S. Stegmaier, M. Magallon, and T. Ertl. A generic solution for hardware-accelerated remote visualization. In *Proceedings of Data Visualisation*, pages 87–95, Barcelona, Spain, 2002.

[21] Helmuth Trefftz, Ivan Marsic, and Michael Zyda. Handling heterogeneity in networked virtual environments. In *Proceedings of IEEE VR'02*, pages 7–15, Orlando, FL, 2002.